

Inductive Types

Part one :

Computation and Deduction
with chunks of Coq in it !!

Benjamin Werner

INRIA-Rocquencourt

Proofs-as-Programs Summer School
Eugene, Oregon, June 27th 2002

Prologue 0 : logical cuts

$$\frac{\frac{\frac{\text{I} \vdash A}{\pi_1}}{\text{I} \vdash A} \quad \frac{\frac{\frac{\text{I} \vdash A \wedge B}{\wedge\text{-!}}}{\text{I} \vdash A}}{\pi_2}}{\wedge\text{-!}}$$

simplifies to

$$\frac{\text{I} \vdash A}{\pi_1}$$

Prologue 1 : numbers in Arithmetic

Axioms :

$$\forall x \quad x = x + 0$$

$$\forall x \forall y \quad (x + y) = y + x$$

$$0 = x * 0$$

$$\forall x \forall y \quad (x * y) = y * x$$

Induction scheme :

$$P(0) \wedge (\forall x \cdot P(x) \rightarrow P(S(x))) \rightarrow \forall n \cdot P(n)$$

Prologue 2 : axiomatic cuts

$$\frac{\frac{P(0)}{\pi_0}}{\frac{\forall n.P(n) \Rightarrow P(S(n))}{\pi_S}}$$

$$\frac{\forall n.P(n)}{P(0)}$$

simplifies to

$$\frac{P(0)}{\pi_0}$$

Real axioms deserve real cuts

Digression : Coq Syntax crash course

$\lambda x : A.t$

$[x:A]t$

$\forall x : A.t$

$(x:A)B$

$\Pi x : A.B$

What is a Type ?

A type is a **Set** :

$\mathbb{N}, \mathbb{N} \rightarrow \mathbb{N}$ are types

$\{n : \mathbb{N} \mid \text{even}(n)\}$ can be a type

$\{0 ; x \mapsto x + 1\}$ is not

A type is a set with a uniformity of

structure. This allows uniform definition of programs.

Concrete Types (in programming)

In ML (SML, Caml...), a concrete type is defined as **closed** by a set of **constructors**.

```
type bool = true | false
```

```
type nat = 0 | S of nat
```

```
let bnot = function true -> false  
          | false -> true
```

Adding concrete types to your logic

Coq syntax :

Inductive bool : Set := true | false | bool

Inductive nat : Set := 0 | nat | S : nat -> nat.

Fixpoint plus [n,m:nat] : nat :=

Cases n of 0 => m

| (S d) => (S (plus d m))

end.

$2+2=4$ is proved by reflexivity !

$$\text{(conv)} \quad \frac{\Gamma \vdash t : A \quad A =_{\beta} B}{\Gamma \vdash t : B}$$

(plus 2 2) = 4 \triangleright 4 = 4.

This extends to **propositions** :

plus is a **program** : (plus 2 2) \triangleright 4

$$\text{(plus } (S \ n) \ m) \triangleright (S \ (\text{plus } n \ m))$$

$$\text{(plus } 0 \ m) \triangleright m$$

New reductions \Rightarrow new proofs !!!

reasoning about inductive types

“Inductive” means the type is the **smallest** type closed under the constructors.

The only canonical objects of type `nat` are `0`, `(s 0)`, `(s (s 0))`, etc

Given any property $P : \text{nat} \rightarrow \text{Prop}$, if :

– $(P\ 0)$ holds,

– $\forall n : \text{nat}. (P\ n) \rightarrow (P\ (S\ n))$,

then $(P\ n)$ holds for any $n : \text{nat}$

\Leftrightarrow induction principle

reasoning about inductive types

For any inductive type definition, we add
an **induction axiom**

`nat_ind`

`: (P : (nat -> Prop))`

`(P 0) ->`

`((n : nat) (P n) -> (P (S n))) ->`

`(n : nat) (P n)`

Same thing for booleans, lists, etc.

structural recursion

Answer : we restrict recursive calls to **structurally smaller** arguments.

Fixpoint plus [n,m:nat] : nat :=

Cases n of 0 => m

| (S d) => (S (plus d m))

end.

positivity condition

Inductive Foo : Set := C : (KXX->Foo)->Foo.

is problematic :

- What should the induction principle be ?
- It breaks the termination property **even**

without Fixpoint command !

in Caml :

```
type foo = C of (foo->foo)
let loop (c f) = f (c f)
```

```
then (loop (c loop)) <sup>14</sup> (loop (c loop))
```

To sum up

To first, or higher, -order logic, we add datatypes à la ML,

with positivity restriction,

with structural recursion restriction,

with a *ad hoc* induction axiom for each definition.

we obtain :

a logic where objects are programs

shorter proofs through computations.

Gödel's system T

It is the core of this calculus.

I.e. simply-typed λ -calculus extended with :

an atomic type nat , $O : \text{nat}$, $S : \text{nat} \rightarrow \text{nat}$

a family of combinators

$R_T : T \rightarrow (\text{nat} \rightarrow T \rightarrow T) \rightarrow \text{nat} \rightarrow T$

reduction rules :

$(R_T t_0 t_S O) \triangleright t_0$

$(R_T t_0 t_S (S n)) \triangleright (t_S n (R_T t_0 t_S n))$

Theorem System T is strongly normalizing
and confluent

Proof : Usual reducibility technique.

Informal : system T morally **is** the system I
described up to here.

Induction Cuts

```
let :  
P:mat->Prop p0:(P 0) ps:(n:mat)(P n)->(P(S n))  
then
```

```
(nat_ind P p0 ps 0) : (P 0) should
```

```
simplify to p0
```

```
(nat_ind P p0 ps (S n)) should simplify to  
(ps n (nat_ind P p0 ps n))
```

Hey! these are exactly the reductions of
system T...

Martin-Löf's Type Theory (1)

Martin-Löf's Type Theory is :

System T enriched with dependent types

or equivalently :

XII enriched with inductive types

New rules :

$\boxed{\vdash \text{nat} : \text{Prop}}$ $\boxed{\vdash O : \text{nat}}$ $\boxed{\vdash S : \text{nat} \rightarrow \text{nat}}$

$\Gamma \vdash T : \text{nat} \rightarrow \text{Prop}$

$\frac{R_T : (P O) \rightarrow (\forall m : \text{nat}. (P m) \rightarrow (P (S m))) \rightarrow \forall n : \text{nat}. (P n)}$

This **dependent** typing of R_T can be obtained by generalizing the typing of pattern matching.

```

P: nat -> Prop
p0 : (P 0)
ps  : (n: nat) (P (S n))
x: nat

```

```

<P>cases x of 0 => p0
| (S n) => (ps n) end : (P x)

```

Inductive definitions of connectors

$$\text{(}\forall\text{-e)} \quad \frac{\Gamma \vdash A \wedge B \quad \Gamma \vdash A \leftrightarrow B \leftrightarrow C}{\Gamma \vdash C}$$

coded as an inductive definition!

Inductive and : Prop := conj : A -> B -> and.

Inductive definition of disjunction

$$\frac{\text{I} \vdash A}{\text{I} \vdash A \vee B} \quad \frac{\text{I} \vdash B}{\text{I} \vdash A \vee B}$$

$$\frac{\text{I} \vdash A \rightarrow C \quad \text{I} \vdash B \rightarrow C}{\text{I} \vdash A \vee B \rightarrow C}$$

Inductive or [A,B:Prop] : Prop :=

left : A -> (or A B)

| right : B -> (or A B) .

The existential quantifier

Martin-Lof presentation :

$$\frac{\Gamma \vdash A : Prop \quad \Gamma, x : A \vdash B : Prop}{\Gamma \vdash \exists x : A.B : Prop}$$

$$\frac{\Gamma \vdash a : A \quad \Gamma \vdash b : B[x \setminus a]}{\Gamma \vdash (a, b) : \exists x : A.B}$$

$$\frac{\Gamma \vdash c : \exists x : A.B \quad \pi_1(c) : A}{\Gamma \vdash c : \exists x : A.B \triangleright a} \quad \frac{\Gamma \vdash c : \exists x : \exists x : A.B \quad \pi_2(c) : B[x \setminus \pi_1(c)]}{\Gamma \vdash c : \exists x : A.B \triangleright b}$$

Retrieving Heyting's semantics

I an inductive type

$$\boxed{\vdash t : I}$$

Then the normal form of t starts with a constructor.

$$I = \Sigma x : A.B \Rightarrow t \triangleright_* (a, b)$$

$$I = A \vee B \Rightarrow t \triangleright_* \text{left}(a) \text{ or } \text{right}(b)$$

Inductive predicates

The smallest set :

– containing 0

– closed by $x \mapsto x + 2$

Inductive even : nat -> Prop :=

E0 : (even 0)

| ES : (n:nat) (even n) -> (even (S n)).

\Leftrightarrow associated induction principle

- Why not the impredicative encoding ?
- induction scheme is not provable in CC
- slow computations
- extraction towards ML
- proving $\perp \neq 0$

Conclusion

Smooth materialization of Curry-Howard

Using Computation in Proofs

Computing with proofs

Powerfull generic mechanism